Logs on Logs on Logs no more: APPEND ATOMIC & REMAP Storage Primitives
Venkatesh Srinivas & Eric Mackay

1. Overview

Storage systems offer various interfaces and guarantees to clients -- file systems offer hierarchically named segments, databases offer sophisticated query languages on indexed data, and block devices (spindles & flash) offer arrays of consecutively-numbered blocks to read to/write from. Each software storage system offers different atomicity and durability properties and depends on other storage systems for providing some service; but each layer implements durability in its own specific way.

File systems & databases consist of sophisticated data structures for their hierarchical name spaces or indexes respectively. Block devices traditionally only provide unordered non-atomic multi-sector writes. File systems & databases implemented atomic updates to their data structures via a number of strategies -- journalling/write ahead logging, undo logging, or copy-on-write. Each of these strategies introduce overheads -- journalling of physical updates requires writing all data twice, halving write bandwidth.

Certain classes of block devices themselves cannot update sectors in place -- commonly used NAND flash implements a traditional block device interface on top of log-structured storage. Future SMR ('shingled') drives will not be able to overwrite random sectors without rewriting bands. Each of these devices duplicate substantial parts of a file system/database's write-ahead logs -- duplication at each layer multiplies overheads while providing no added atomicity or durability properties.

System efficiencies would improve if block device interfaces were improved, to expose atomic write capabilities to users.

2. WRITE ATOMIC

Recently SCSI / T10 added a 'WRITE ATOMIC' command. WRITE ATOMIC updates a contiguous range of sectors; the multi-sector write commits to stable storage whole or does not commit at all. This primitive is very easy for log-structured storage to implement -- it is no more complex than a non-atomic write for NAND flash. Write-ahead logging strategies in SQLite or the ext4 file system, for example, could use WRITE ATOMIC to append multi-sector updates to their logs and save the cost of writing a commit block after a log append.

Unfortunately SCSI / T10 did not implement a 'scattered' atomic write -- an atomic write that can update multiple non-contiguous sector ranges. Scattered Write Atomic was proposed but not ratified.

Write Atomic as written in the SCSI/T10 standards has some downsides for our applications. Namely, it cannot perform writes to more than one location; all data is written contiguously. Since most file systems & databases require writes to multiple noncontiguous LBAs as a single atomic operation in order to maintain consistency, this makes Write Atomic a poor choice for replacing database logs.

A first strategy would be to implement scattered write atomic, from the T10 draft [1]; prior literature assumes some form of scattered atomic write interface, see [2] for example.

We observe that a write (even a scattered atomic write) to log-structured storage represents two operations -- 1) appending data atomically to media; 2) updating translation tables atomically.
We propose a different strategy than T10 -- expose two commands, to represent the two independent operations storage devices provide:

4. APPEND_ATOMIC and REMAP

4.1 APPEND_ATOMIC has similar command structure to a write operation. It takes the data we wish to write, but not a Logical Block Address (LBA) like we would expect in a regular write operation. It returns a

token, also known as a Representation of Data (ROD), that uniquely identifies the written data and what physical block address it was written at. Several APPEND_ATOMIC operations can proceed in parallel, up to the limit of the storage device. The APPEND_ATOMIC operation does not provide ACID guarantees itself, but is meant to be used in conjunction with the REMAP operation.

4.2 REMAP takes a list of (LBA, ROD, length) triples. It changes the mapping of the given LBA in the flash translation layer to point to whatever PBA the ROD corresponds to. The length indicates the number of blocks that contain the data that was written. The number of triples that a single REMAP operation can remap is the limit of the ACID granularity that we can support. The REMAP operation also unmaps the physical blocks that the LBA previously pointed to.

We expect to see several APPEND_ATOMIC operations followed by a REMAP operation. The writes performed by the APPEND_ATOMIC are not visible or durable until after the REMAP completes. There is no way to access the data written because no LBA points to it. If the storage device has a failure before the REMAP completes, we still recover in a consistent state because it appears as if the APPEND_ATOMIC operations never happened. This can greatly simplify Database Management Systems (DBMS) because there is no longer a need to provide ACID guarantees in software through techniques like logging; the storage device provides these capabilities. We expect to see significantly improved performance because of several factors:

- Eliminating the need for logging and other ACID techniques that are often duplicated in several layers of the system stack
- Write-ahead logging requires 2 actual write operations to make an ACID write. This decreases the useful lifetime of storage devices that wear out, such as Flash. This also uses up bandwidth, and only half of the bandwidth is being used for actual work.
- Write operations are often serialized when they don't need to be. APPEND_ATOMIC operations can be performed concurrently and the only bottleneck is the single REMAP operation afterwards.

4.3 Instruction Formats

Byte\Bit	7	6	5	4	3	2	1	0
0	Operation Code (3Dh)							
1	Reserved	Reserved	Reserved	DPO	FUA	Reserved	FUA_NV	Reserved
2	(Reserved, must be zero)							
3	(Reserved, must be zero)							
4	(Reserved, must be zero)							
5	(Reserved, must be zero)							
6	Allocation length -- number of bytes reserved in the data-in buffer							
7	(MSB) Transfer Length, in blocks; number of blocks to write to the device.							
8	Transfer length, in blocks (LSB)							
9	Control							

In the data in buffer, the SCSI target writes back a structure in response to the APPEND ATOMIC command. The structure serves as a 'representation of data', a token to use for REMAPs.

Byte\Bit	7	6	5	4	3	2	1	0
0	(MSB) ROD token							
1	ROD token							
2	ROD token							
3	ROD token							
4	ROD token							
5	ROD token							
6	ROD token							
7	ROD token (LSB)							

SCSI REMAP

The REMAP command takes a 10-byte CDB. The structure is very similar to UNMAP.

Byte\Bit	7	6	5	4	3	2	1	0
0	Operation Code (49h)							
1	Reserved, MBZ							
2	Reserved, MBZ							
3	Reserved, MBZ							
4	Reserved, MBZ							
5	Reserved, MBZ							
6	Reserved, MBZ							
7	(MSB) Parameter list length, in bytes.							
8	Parameter list length, in bytes (LSB)							
9	Control							

Along with a REMAP command, a parameter list is transferred. It is in the data out buffer, it is just a concatenation of REMAP descriptors with LBAs.

Byte\Bit	7	6	5	4	3	2	1	0
0	(MSB) Logical Block Address							
1	Logical Block Address							
2	Logical Block Address							
3	Logical Block Address							
4	Logical Block Address							
5	Logical Block Address							
6	Logical Block Address							
7	Logical Block Address (LSB)							
8	(MSB) ROD token							
9	ROD token							
10	ROD token							
11	ROD token							
12	ROD token							
13	ROD token							
14	ROD token							
15	ROD token (LSB)							
16	(MSB) Length of contiguous data to remap, in blocks							
17	Length of contiguous data to remap, in blocks							
18	Length of contiguous data to remap, in blocks							
19	Length of contiguous data to remap, in blocks (LSB)							

5. Implementation

We implemented our custom SCSI commands in a userspace iSCSI target, stgt. The userspace iSCSI target exports block devices from files on a hosting system and handles SCSI protocol decode/framing.
In our test environment, we used files on tmpfs (in RAM) to back our block devices.

We then connected to the custom iSCSI target from a Linux virtual machine running on the same host; the virtual machine guest issued custom commands using the Linux sg (SCSI generic) interface. Userspace programs (sqlite3 in particular) can issue SCSI-generic commands, allowing us to test on unmodified Linux kernels.

We ran tests on a machine with an AMD 6-core Piledriver CPU, 8 GB of DRAM, and a single local SSD for storage; the local SSD was limited to 40k Read IOPS and 15k Write IOPS at 4 KB I/Os. We used Ubuntu 15.04 as the hosting OS and QEMU 2.3 as the virtual machine monitor.

Real FTLs on modern flash storage map a logical block address space into a larger physical block address space; the physical block space is overprovisioned to allow writes to avoid read-modify-write (program-erase) cycles. After writes have written into overprovisioned space, the FTL maps are updated to point logical blocks at the newly written area. Our proposed commands split the write-in-unallocated-space and remap operations of existing FTLs; we need a similar underlying storage architecture (overprovisioned, unaddressable space) to implement our commands.

In our prototype, we map a logical block device onto a simple UNIX file via a linear mapping. We introduce a secondary file, treated as an append-only store. APPEND ATOMIC writes into the append-only store rather than to the underlying file. We also introduce an in-memory data structure called a ‘diversion table’ -- the diversion table maps logical blocks onto locations in the append-only store. If a logical block address is not in the diversion table, it is read from the simple linear file. Multi-sector reads are splintered into single-block reads from the appropriate backing store (a multi-sector read can touch a mix of blocks from both the linear file and the append-only file); this introduces an overhead for multi-sector reads relative to an unmodified system. The implementation could definitely be improved, to service multi-sector reads in contiguous ranges, but we have not done so. This implementation is also unrealistic in that the append-only store grows without bound (as a function of the APPEND ATOMIC rate). The code is structured so that it would be easy to use *fallocate(FALLOC_FL_PUNCH_HOLE)* to release unused blocks in both the linear file and the append log, but we have also not done so.

In our prototype, REMAP inserts point ‘diversions’ into the diversion table; the table is implemented on a simple C++ STL map<>. We could improve efficiency for multi-sector writes by either using an interval tree (rather than a point-mapping tree) and/or using multilayer bitmaps, just as some real FTLs do.

We then patched sqlite3 in Write-Ahead Logging mode to use the new SCSI commands when applicable -- at the entry point of the SQLite3 Write-Ahead-Log, we have insight into the final locations of all data we want to update. We do not generate the WAL entry; we instead issue a series of APPEND ATOMIC commands to write the updated pages out. When we exit the WAL code, back to the location where we would do in-place writes, we switch to issuing a REMAP command.

6. Evaluation & Results

We ran the sqlite3 benchmark that is provided with the well-known LevelDB key-value store. The benchmark only uses SQLite3 as a key-value store and does not take advantage of other properties (sophisticated queries, cross-table constraints, etc); however it does exercise the I/O paths of SQLite.
The benchmark creates entries, and then performs several sets of operations such as overwriting values, sequential writes, random writes, sequential reads, random reads, and overwrites.

We ran the benchmark with both unpatched and patched SQLite3 libraries, with 8 kB and 163 kB values (16-byte keys). Both test configurations ran against our modified iSCSI target, to isolate the extra overhead of block-reads from the new commands. We ran the tests on both ext2 and ext4 (with journaling disabled) filesystems per recommendations on the LevelDB mailing list.

The LevelDB benchmarks report microseconds per operation for every test and throughputs from some tests. We focus on microseconds/operation -- at fixed concurrency there is a linear relationship between 1/uSec and throughput.

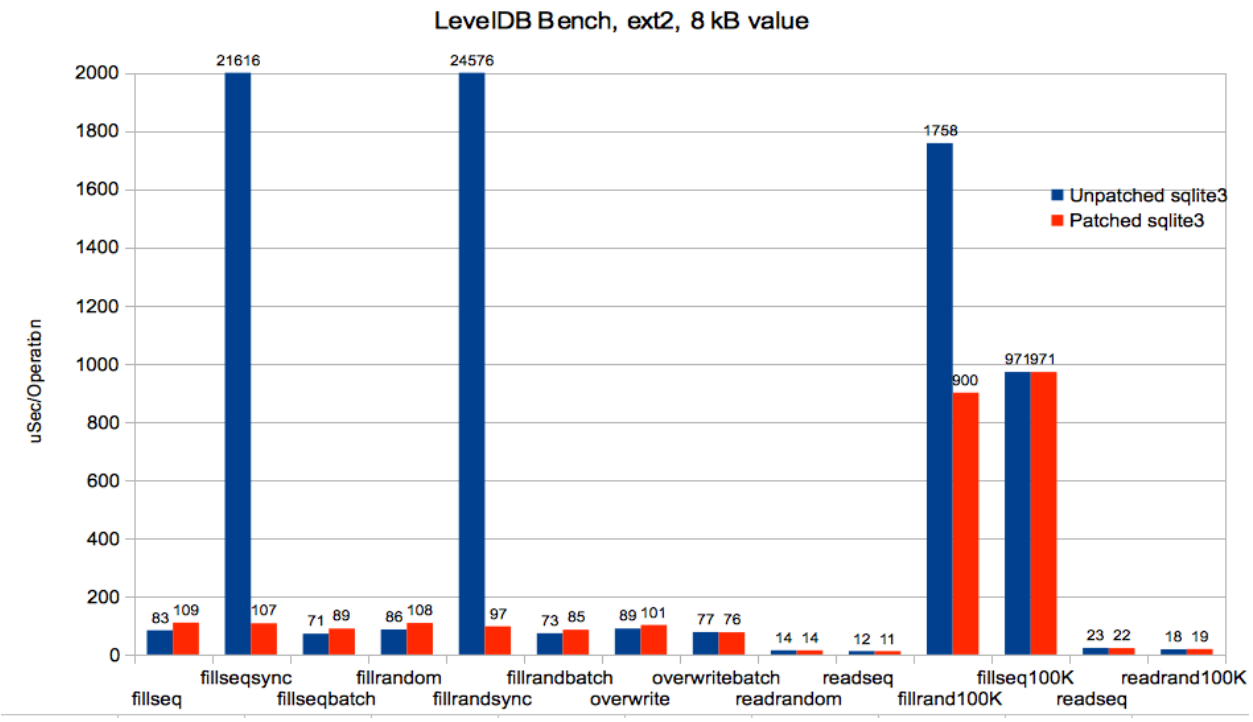

Figure 1: 8 kB values with sqlite3/ext4

First, we look at LevelDB bench with 8 kB keys on ext2. Of note -- the plot's y-axis is truncated. Point-Synchronous operations in an unpatched sqlite3 database are very slow and would make it impossible to see differences in other tests on the same plots.

In a number of sequential key-fill tests (fillseq, fillseqbatch, fillrandom, fillrandbatch, overwrite), we see the unmodified sqlite outperform our patched sqlite database. We suspect this arises for two reasons -- for sequential fills, sqlite3 should be able to issue minimally scattered I/Os. The costs associated with maintaining our diversion table are likely outweighing the win from reduced I/Os.

In one test, fillrand100K (write 10,000 100K values in random order in async mode), we see a very solid speedup -- uSec/Op is halved. This is consistent with our expectations; with APPEND ATOMIC/REMAP, we expect to write half the data that SQLite3's WAL mode would write. The random write test benefits, but the fillseq100K test does not; we believe this is because SQLite3 or the rest of the storage stack are able to merge I/Os to contiguous sectors. Our APPEND ATOMIC is more of a win when scattered I/Os are in play.

We have a hypothesis as to why synchronous key inserts are so slow on unpatched SQLite3 -- the write/commit sequence for synchronous key updates on sqlite3 + a Write Ahead Log:

1) Takes a WAL checkpoint (including a full fsync)
2) Writes the single transaction (to the WAL, then to the final location).
3) Issues another fsync & waits.

Our iSCSI target uses *tmpfs* as its backing store; fsync is basically free. However, the window in which only one database transaction is in-flight is still a liability. Even if fsync were a no-op, going to a single-transaction state is very expensive.

With APPEND ATOMIC/REMAP, we can allow independent I/O to proceed; the wait would only happen at REMAP time.

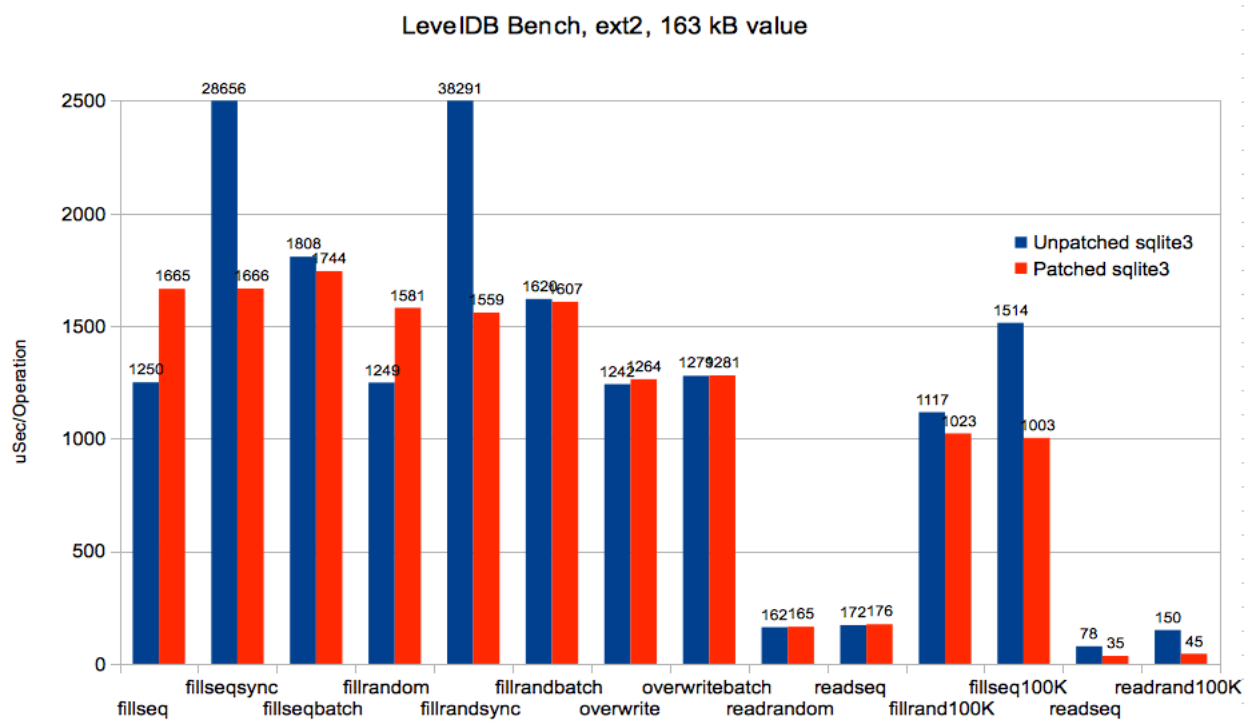

Figure 2: 163 kB values with ext2

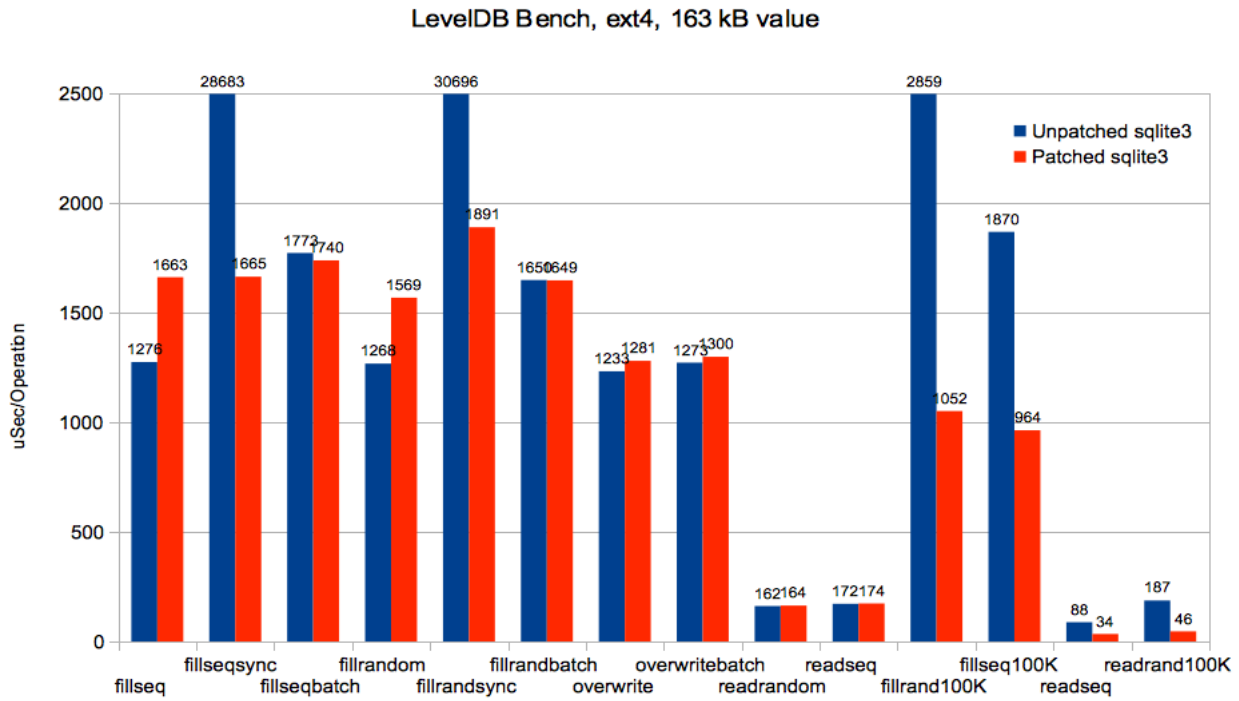

Figure 3: 163 kB values with ext4

We next ran the LevelDB benchmark with 163 kB values instead of the 8 kB values used in the first test. With much larger keys, we expect:

1) Bulk I/O costs would become more apparent.
2) Per-request overheads would become less significant.

We first observe in fillseq test on both ext2 and ext4 that a patched SQLite3 underperforms an unpatched sqlite3, by a larger margin than with 8 kB keys. We believe this is due to diversion table overheads -- a single 163 kB value would require up to 326 point-mapping entries in the diversion table. 10,000 keys would result in a fairly large/deep map data structure.

We see a similar difference in fillrandom tests; we believe this is for the same reasons.

We still see very long synchronous I/O times with an unpatched build of SQLite3 and 163 kB I/Os. Of interest is that the uSec/Operation for synchronous I/Os in an unmodified SQLite3 are not very different for 8 kB or 163 kB I/Os. This is consistent with our theory that the window of time when only one I/O is in-progress is the performance limiter for synchronous updates.

7. Citations:

1. Curtis Ballard. “DRAFT: 15-087r1 SBC-4 SPC-5 Scattered writes”. T10 Technical Committee, 4/21/2015
2. Xiangyong Ouyang; Nellans, D.; Wipfel, R.; Flynn, D.; Panda, D.K., "Beyond block I/O: Rethinking traditional storage primitives," *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* , vol., no., pp.301,311, 12-16 Feb. 2011 doi: 10.1109/HPCA.2011.5749738